VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komprese metodou ACB

ACB Compression Algorithm

2018 Radek Huráb

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Radek Huráb**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Komprese metodou ACB
ACB Compression Algorithm

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout efektivní implementaci kompresní metody Associative Coder of Buyanovsky. Práce bude realizována v jazyce C++ nebo C# a budou použity jen vlastní datové struktury vybrané pro rychlou a efektivní práci algoritmu. Bude možné měnit parametry algoritmu.

Práce bude obsahovat:
1. Podrobný popis algoritmu ACB.
2. Návrh datových struktur pro realizaci jednotlivých částí algoritmu.
3. Implementaci algoritmu ACB.
4. Otestování algoritmu nad různými testovacími soubory a jeho porovnání s jinými algoritmy.
5. Návrh možných modifikací algoritmu a jejich případná realizace.

Výsledná aplikace bude ve formě konzolové aplikace a všechny možnosti bude možné nastavit pomocí parametrů pro příkazovou řádku.

Seznam doporučené odborné literatury:

[1] Salomon D., Data Compression: The Complete Reference, Springer, 2004, ISBN 978-038-740-6978
[2] Valach Michal, Efficient implementation of ACB compression algorithm for ExCom library, 2011, Dipl. práce CVUT, FEL)

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.	prof. Ing. Pavel Brandštetter, CSc.
vedoucí katedry	*děkan fakulty*

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 20. dubna 2018 .....

Rád bych poděkoval panu doc. Ing. Janu Platošovi, Ph.D. jakožto vedoucímu této bakalářské práce, který nikdy neodmítl moje otázky nebo prosbu o konzultace, které mi pomohly při vypracování této práce.

Abstrakt

Tato práce se zabývá kompresní metodou ACB, její implementací a návrhem struktur pro realizaci jednotlivých částí algoritmu s provedením následného měření na testovacích souborech. Navrhuje možné modifikace tohoto algoritmu a také větší část z nich realizuje do jednotlivých metod programu. Výsledky takto modifikovaných metod jsou poté zahrnuty v měření.

Klíčová slova: komprese, dekomprese, ACB

Abstract

This work deals with ACB compression method, its implementation and design of structures for realization of individual parts of algorithm, followed by measurements on test files. It proposes possible modifications of this algorithm and also implements a larger part of the algorithms in individual program methods. The results of these modified methods are then included in the measurements.

Key Words: compression, decompression, ACB

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam tabulek **10**

Seznam výpisů zdrojového kódu **11**

1 Úvod **12**
1.1 Historie . . . 12
1.2 Motivace . . . 12
1.3 Hlavní cíle práce . . . 12
1.4 Organizace textu . . . 13

2 Datová komprese **14**
2.1 Úvod do datové komprese . . . 14
2.2 Pojmy a definice . . . 14
2.3 Entropie a redundance . . . 15
2.4 Klasifikace komprese . . . 15

3 Algoritmus ACB **17**
3.1 Komprese ACB . . . 18
3.2 Dekomprese ACB . . . 22
3.3 Modifikace algoritmu . . . 25

4 Implementace algoritmu ACB **27**
4.1 Datové struktury . . . 27
4.2 Kódování výstupu . . . 29
4.3 Modifikace . . . 30
4.4 Parametry metody . . . 30
4.5 Diagram tříd . . . 31
4.6 Použití . . . 33

5 Výsledky měření implementovaným algoritmem **35**
5.1 Vliv základních parametrů na kvalitu komprese . . . 35
5.2 Rozložení četnosti výskytu offsetů a délky . . . 39
5.3 Vliv modifikací na kvalitu komprese . . . 41

6 Závěr **47**

Literatura **48**

Přílohy **48**

A Seznam testovacích souborů **49**

B Použitý rádius k dosažení nejlepší komprese pro daný soubor **50**

C Obsah přiloženého CD **51**

Seznam použitých zkratek a symbolů

ACB	–	Associative Coder of Buyanovsky
LZ77	–	Lempel-Ziv kompresní metoda z roku 1977
LZ78	–	Lempel-Ziv kompresní metoda z roku 1978
RAR	–	Roshal ARchive
bps	–	bits per symbol

Seznam obrázků

1 Posuvné okno 17
2 Jednoduché znázornění průběhu veřejných metod třídy *ACB* 29
3 Diagram tříd 32
4 Závislost bps na velikosti *kontextu* 36
5 Závislost rychlosti komprese na velikosti *kontextu* 36
6 Závislost bps na velikosti hodnoty *content* 37
7 Závislost rychlosti komprese na velikosti hodnoty *content* 37
8 Závislost váženého průměru bps na velikosti *rádia* 38
9 Závislost rychlosti komprese na velikosti *rádia* 38
10 Statistika četnosti hodnot offsetů u souboru book1 40
11 Statistika četnosti hodnot *délky* u souboru book1 40
12 Závislost váženého průměru bps na zvolené metodě 45

Seznam tabulek

1 Obsah ACB slovníku 17
2 Rozložení četnosti hodnot *délky* a *offsetu* 40
3 bps vzhledem k použité metodě 43
4 Časy komprese vzhledem k použité metodě 44
5 Průměrná velikost použité hodnoty rádia vzhledem k metodě. 44
6 Časy dekomprese vzhledem k použité metodě 45
7 Použitý *rádius* k dosažení nejlepší komprese pro daný soubor 50

Seznam výpisů zdrojového kódu

1 Úvod

1.1 Historie

Informace jsou v moderní lidské společnosti nedílnou součástí každodenního života, zvláště v posledních letech, kdy objem informací, ať už ukládaných či přenášených, stále narůstá. Z tohoto důvodu vzniká větší nárok na fyzická média, jako jsou pevné disky nebo případně datové proudy, které se využívají pro přenos těchto dat. Tuto situaci můžeme řešit zvětšováním kapacity disků nebo zvyšováním propustnosti datových proudů. Dalším řešením je snížení nároku na tyto média za pomocí komprese.

Časným příkladem datové komprese je Morseova abeceda, vyvinutá v roce 1838 pro použití v telegrafii. Moderní komprese dat má počátky na začátku padesátých let dvacátého století s vývojem informačních technologií popsaných v knize "*New kind of science: notes from the book*" [3]. Nejpopulárnějšími metodami se staly LZ77 a LZ78, které byly popsány na konci sedmdesátých let publikované A. Lempelem a J. Zivem v článcích [1, 2]. Tyto varianty se staly základem mnoha jiných kompresních algoritmů.

1.2 Motivace

V roce 1994 byla publikována kniha "*Associative Coding*" [4] Georgem Buyanovskym v ruském jazyce, která popisovala novou metodu komprese dat nazývanou ACB (Associative Coder of Buyanovsky), která se stala velmi efektivní bezeztrátovou metodou komprese, ale navzdory své efektivitě byla oproti svým předchůdcům pomalá. Příčina je v neustálém přeuspořádání slovníku, který je identický jak při kompresi tak dekompresi.

Dokumentace této metody je ovšem k dnešnímu datu zastaralá a implementace tohoto algoritmu, která nebyla nikdy zveřejněna, je chráněna patentem.

1.3 Hlavní cíle práce

Cílem této práce je navrhnout efektivní implementaci kompresní metody ACB realizovanou v jazyce C# za použití vlastních datových struktur vybraných pro rychlou a efektivní práci algoritmu. První výzvou práce je navržení a implementace datové struktury, která bude efektivní pro vyhledávání, vkládání a mazání vytvářených dat. Dalšími výzvami bude práce s výstupními daty komprese, jejich následné zakódování do výsledného souboru a návrh možných modifikací algoritmu s jejich případnou realizací.

1.4 Organizace textu

Kapitola první se věnuje historii vzniku komprese, motivaci a hlavním cílům této práce.

Druhá kapitola se snaží nahlédnout do základních pojmů a definicí související s datovou kompresí. Snaží se zdůraznit a popsat informace, které jsou obsaženy či použity v této práci pro lepší pochopení celého problému.

Kapitola třetí se obecně zabývá algoritmem ACB. Popisuje postup při kompresi a dekompresi s následnou ukázkou pomocí příkladů. Dále navrhuje a popisuje možné modifikace tohoto algoritmu.

Čtvrtá kapitola se věnuje implementaci tohoto algoritmu. Popisuje strukturu a metody jednotlivých tříd, které jsou poté využívány pro kompresi a dekompresi dat. V této kapitole je obsažen diagram tříd implementovaného algoritmu a popis pro použití výsledného programu. Dále je zde popsána implementace modifikací zmíněných v předešlé kapitole.

Pátá kapitola hodnotí vliv parametrů na kvalitu komprese naměřenou na zkušebních souborech. Také porovnává implementace různých modifikací daného algoritmu a jejich vliv na výslednou kvalitu komprese.

Šestá závěrečná kapitola obsahuje shrnutí dosažených výsledků této práce a také možné návrhy pro budoucí zlepšení či rady pro další výzkum na toto téma.

2 Datová komprese

2.1 Úvod do datové komprese

Komprese zpracovává data do takové podoby, aby zmenšila jejich objem a přitom zanechala informace v nich obsažené. Jejím cílem je zmenšit zatížení zdrojů při ukládání informací nebo zmenšit nápor na datové toky, které tato data přenáší. Pomocí určitého algoritmu můžeme data kódovat tak, aby z nich byly odstraněny informace, které jsou redundantní neboli nadbytečné. Opakem komprese nazýváme dekompresi.

2.2 Pojmy a definice

Posuvné okno - Je rozděleno na dvě části, které obsahují výřez ze zpracovávaných dat. První, levá část, obsahuje již zpracovaná data a druhá, pravá část obsahuje data, která teprve budou zpracována.
Kontext - Jedná se o množinu *symbolů* stvořených z levé části okna.
Aktuální kontext - Jedná se o *kontext*, který se momentálně nachází v levé části posuvného okna, jehož obsahem jsou zakódovaná data.
Content - Vzhledem k tomu, že anglické slovo "content" nemá v češtině dostačující jednoslovný výraz, je v této práci použit jako označení pro množinu *symbolů*, které jsou stvořeny z pravé části okna.
Aktuální content - Jedná se o *content*, který se momentálně nachází v pravé části posuvného okna. Data obsažená v hodnotě *content* je potřeba zakódovat.
Korpus - Je sada různých souborů používaných k vyhodnocení kompresních metod.
Datové proudy - Tyto proudy jsou sekvence dat. Datový proud je definován svým vstupem a výstupem.
Komprese - Komprese neboli komprimace je zpracování počítačových dat s cílem zmenšit jejich objem se současným zachováním informací obsažených v datech.
Dekomprese - Dekomprese neboli dekomprimace je opačný proces komprese, kdy z komprimovaných dat vytváříme data původní.
Kompresní poměr - Je poměr velikosti komprimovaných dat vzhledem k velikosti dat původních.

$$\text{kompresní poměr} = \frac{\text{velikost komprimovaného souboru}}{\text{velikost původního souboru}} \tag{1}$$

bps - Jedná se o jiný způsob vyjádření kompresního poměru, který je dán počtem bitů na *symbol*. Udává tedy kolik je v průměru potřeba bitů pro uložení jednoho bajtu výchozího souboru během komprese.

$$\text{bps} = 8 * \text{kompresní poměr} \tag{2}$$

Délka - Je označení pro přirozené nezáporné číslo, udávající počet shodných *symbolů*.

Offset - Jedná se o indikátor vzdálenosti v kolekci. Číslo ve formátu integer je zde nazýváno *offset*, který definuje pozici určitého prvku v objektu (kolekci) neboli jeho vzdálenost od počátku objektu.
Rádius - Jedná se o poloměr vyhledávání využívaný v implementovaných metodách. Jeho hodnota, udávaná v bitech, reprezentuje maximální velikost *offsetu*.
Položka slovníku - Tento prvek definuje kontext a *content*, které jsou definovány množinou *symbolů*.
Slovník - Jedná se o uspořádanou skupinu elementů (položek).
Symbol - *Symbolem* je nazýván jeden samostatný prvek reprezentovaný jedním bajtem.
Trojice - Je složena ze tří prvků a to z *offsetu*, *délky* a *symbolu*.

2.3 Entropie a redundance

Entropie (informační entropie) je také často nazývána Shannonovou entropií po Claudovi E. Shannonovi. Tento americký elektronik a matematik zformuloval mnoho klíčových poznatků teoretické informatiky. Entropie je definována jako průměrná hodnota informace obsažené ve zdrojových datech. Obecně řečeno, většina dat obsahuje nějakou nadbytečnou informaci.

Redundance porovnává aktuální entropie těchto informací s teoretickým maximem entropie. Čím vyšší je redundance, tím více dané informace může být komprimováno.

2.4 Klasifikace komprese

Komprese se dělí na bezeztrátovou a ztrátovou. Bezeztrátová komprese umožňuje perfektní rekonstrukci originálních dat z dat komprimovaných, nejčastěji v případech, kdy nelze dopustit ztrátu informace, jako je tomu tak například v textových souborech. Naproti tomu ztrátová komprese, která má však mnohem vyšší kompresní poměr, rekonstruuje pouze hlavní složku originálních dat a lze ji použít pouze v případech, kdy je ztráta akceptovatelná, například u komprese obrazu či zvuku. Komprese se dále dělí podle metody komprese.

První metodou kompresního algoritmu je statistická metoda. Princip statistických metod spočívá na frekvenci výskytu každého *symbolu* ve vstupních datech. *Symbolům*, jež se vyskytují častěji ve zdrojových datech, je přidělena kratší bitová reprezentace a *symbolům*, mající naopak malý výskyt, je přidělena delší bitová reprezentace.

Druhou metodou kompresního algoritmu je slovníková metoda, která je založená na principu vyhledávání spojitostí mezi vstupní množinou *symbolů* a množinou už zpracovaných dat, které jsou v datové struktuře (nazývané slovník) obsluhované kodérem. Pokud kodér nalezne takovouto shodu, nahradí danou množinu referencí na příslušnou položku v slovníku.

Více informací o dalších metodách můžeme nalézt v dokumentu "*Analysis of Compression Algorithms for Program Data*" [6].

2.4.1 Huffmanovo kódování

Tento algoritmus, jehož autorem je David A. Huffman, byl publikován roku 1952 v článku "*A Method for the Construction of Minimum-Redundancy Codes*" [7]. Huffmanovo kódování se využívá v bezeztrátové kompresi a jedná se o široce využívanou statistickou metodu komprese. Algoritmus během kódování vytvoří pole ze všech vstupních *symbolů*, setříděné podle jejich výskytu. Poté z tohoto pole vytvoří binární strom, kde každý list tohoto stromu představuje jeden ze *symbolů*. V tomto stromu projde každou jeho část, aby určil jednotlivé kódy pro každý list (*symbol*). *Symboly* s nejvyšším výskytem mají kratší kód. Během dekódování se jednoduše vykonává opačný proces: kód je využit k průchodu stromem k určitému *symbolu*.

3 Algoritmus ACB

ACB je bezeztrátový kompresní algoritmus používající slovníkovou metodu. Jeho zaměření je hlavně na textové soubory, kde využívá toho, že se v textu nevyskytují jednotlivá slova nahodile, ale že mezi nimi existuje určitá souvislost (*kontext*). Většina této kapitoly odkazuje na popis algoritmu ACB publikovaný v knize "*Data Compression: The Complete Reference*" od Davida Salomona, [5].

Vstupem algoritmu jsou data ve formě *symbolů*, které jsou procházeny postupně od začátku do konce. Při procházení množiny těchto vstupních *symbolů* se kodér snaží najít souvislost mezi právě zpracovávanými daty a daty, které již prošly kompresí a jsou uložené ve slovníku.

Tento algoritmus používá takzvané posuvné okno (Obrázek 1) jako algoritmus LZ77. Levá část posuvného okna je v této práci nazvána *kontext*, a pravá část *content*.

aa|cabaaa|ababcaba|acaba

Obrázek 1: Posuvné okno

Každým cyklem algoritmu je vytvořena jedna a více dvojic *kontextů* a hodnot *content*, které jsou následně ukládány a řazeny lexikograficky do slovníku od posledního písmena jejich *kontextu*. Pokud si vezmeme Obrázek 1 jako příklad momentální pozice okna během kódování, bude stav slovníku identický s tabulkou 1. Během kódování i dekódování musí být stav slovníku identický v každém cyklu.

Tabulka 1: Obsah ACB slovníku

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	acabaa	aababcab
4	aacaba	aaababca
5	aaca	baaaabab
6	aacab	aaaababc
7	aac	abaaaaba

Komprimovaná data jsou poté složeny z trojic **(o, l, s)**.

- **o (*offset*)** - vzdálenost nejvíce podobné hodnoty *content* od vybraného *kontextu*
- **l (délka)** - počet shodných *symbolů* z vybrané hodnoty *content*
- **s (symbol)** - první neshodující se *symbol*

3.1 Komprese ACB

Algoritmus ACB komprese obsahuje několik kroků, které si můžeme vysvětlit za použití pseudokódu, zobrazeného v Algoritmu 1.

Algoritmus 1 ACB komprese

```
1:  procedure ACBCompress
2:      windowPosition ← 0                          ▷ nastavení pozice posuvného okna
3:      while windowPosition < InputSize do
4:          i ← pozice nejvíce se shodujícího kontextu ve slovníku s aktuálním kontextem
5:          j ← pozice nejvíce se shodující hodnoty content ve slovníku s aktuální hodnotou content
6:          l ← počet shodných symbolů nalezené hodnoty content s aktuální hodnotou content
7:          s ← symbol na pozici l + 1
8:          Output(j − i, l, s)
9:          UpdateDictionary()                      ▷ aktualizuj slovník (vlož nové dvojice)
10:         windowPosition ← windowPosition + l + 1
11:     end while
12: end procedure
```

Algoritmus má na počátku prázdný slovník. V první fázi se ve slovníku nalezne nejvíce shodný *kontext* pro aktuální *kontext* a jeho index (i) je zapamatován. Díky tomu, že je slovník seřazen od posledního písmena *kontextu*, může být vyhledávání zajištěno pomocí binárního stromu místo lineárního procházení všemi prvky slovníku. *Symboly*, které jsou obsaženy v aktuálním *kontextu*, už byly kódovány. Dalším krokem je vyhledání indexu nejshodnější hodnoty *content* (j) pro aktuální *content*, který teprve čeká na zakódování. Nejvíce shodný *content* je už hledán pomocí lineárního vyhledávání, pokud je takovýchto položek nalezeno více, má přednost ta, která je nalezena nejblíže k nalezenému *kontextu* (i). Použití lineárního vyhledávání má dopad na výpočetní rychlost, která klesá se zvyšující se velikostí slovníku. Spolu s indexem nalezené hodnoty *content* je i zjištěn počet shodných *symbolů* (*délky*) s aktuální hodnotou *content*.

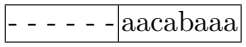
V další fázi se tato kódovaná data mohou poslat na výstup. Jednotlivá data obsahují tři části, nazývané *trojice*. První částí *trojice* je vzdálenost od nejvíce se shodujícího *kontextu* (i) a nejvíce se shodující hodnoty *content* (j). Druhou částí je počet shodných *symbolů* hodnoty *content* (*délka*) a poslední částí *trojice* je první neshodující se *symbol* aktuální hodnoty *content* (s).

Po zapsání této trojice na výstup je třeba aktualizovat slovník. Do slovníku bude přidáno *délka*+1 nových dvojic *kontextů* a hodnot *content*, které vznikly během tvoření dané trojice. Poté se posuvné okno posune na další pozici.

Ukázka komprese, pro lepší pochopení, bude dále uvedena v příkladu 1.

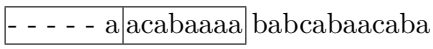
Příklad 1 (Komprese ACB)
Vstupním řetězcem bude text $T = aacabaaaababcabaacaba$. Velikost *kontextu* je nastavena na hodnotu 6 a velikost hodnoty *content* na hodnotu 8. Hranice posuvného okna je na začátku nastavena

tak, aby zpracovaná část textu ležela v levé části okna (*kontext*) a nezpracovaná část textu ležela v pravé části okna (*content*)



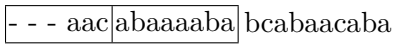
Z důvodu, že je slovník prázdný, není třeba v prvním kroku nic vyhledávat. První *trojicí* výstupu je tedy **(0, 0, a)**. Slovník bude aktualizován o *délka*+1 položek, tedy pouze o jednu položku. Posuvné okno se posune také o jednu pozici doprava.

index	*Kontext*	*Content*
0		aacabaaa



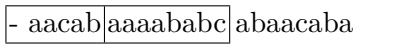
Aktuálnímu *kontextu* odpovídá *kontext* ve slovníku na indexu 0. Nejvíce se shodující *content* je nalezen také na indexu 0 a to se shodou prefixu jednoho *symbolu*. Další kódovanou trojicí bude (0 - 0 = **0, 1, c**). Slovník bude aktualizován o *délka*+1 položek, tedy o dvě položky. Posuvné okno se posune také o dvě pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab



Nejvíce shodný *kontext* ve slovníku je nalezen na indexu 2. Shodný *content* je nalezen na indexu 1 se shodou prefixu 1. Výsledná trojice bude tedy vypadat (1 - 2 = **-1, 1, b**). Slovník bude aktualizován o dvě položky a okno posunuto o dvě pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	aaca	baaaabab
4	aac	abaaaaba



Aktuálním *kontextem* je teď řetězec *aacab*, vzhledem k tomu, že aktuálnímu *kontextu* neodpovídá žádný *kontext* ve slovníku, tak je v takovém případě nalezen nejpodobnější *kontext*, volba spadá mezi třetí a čtvrtý index. Je ponecháno na implementaci, která z položek bude vybrána, ovšem stejný způsob musí být použit i při dekompresi. Bude vybrán *kontext*, který je výše tedy

na indexu 3. Shodný *content* je nalezen na indexu 0 s počtem shodných *symbolů* prefixu 2. Na výstup bude poslána trojice (0 - 3 = **-3, 2, a**). Slovník bude aktualizován o tři prvky a okno posunuto o tři pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	acabaa	aababcab
4	aacaba	aaababca
5	aaca	baaaabab
6	aacab	aaaababc
7	aac	abaaaaba

aa|cabaaa|ababcaba|acaba

Pro aktuální *kontext cabaaa* je vybrán *kontext* ze slovníku na indexu 2 a pro aktuální *content ababcaba* je nalezen *content* ze slovníku na indexu 7 se třemi shodnými *symboly* prefixu. Výsledná trojice tedy bude (7 - 2 = **5, 3, b**). Slovník bude aktualizován o čtyři prvky a okno posunuto o čtyři pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	abaaaa	babcabaa
4	cabaaa	ababcaba
5	acabaa	aababcab
6	aaaaba	bcabaaca
7	aacaba	aaababca
8	aaca	baaaabab
9	baaaab	abcabaac
10	aacab	aaaababc
11	aac	abaaaaba

aacaba|aaabab|cabaacab|a

Při vyhledávání aktuálního *kontextu aaabab* v abecedně uspořádaném slovníku (zprava doleva) je nalezena shoda na indexu 9 a 10, tyto položky jsou pro aktuální *kontext* stejně vhodné. Stejně jako v minulých krocích v případě shody u dvou položek, je vybrána ta, která je výše, tedy na indexu 9. Aktuální *content* se shoduje s hodnotou *content* ze slovníku na indexu 2 se shodou v pěti *symbolech*. Kódovaná trojice bude pro tento krok (2 - 9 = **-7, 5, c**). Slovník bude aktualizován o hodnotu *délky* + 1 položek neboli 6, se současným posunutím okna o 6 pozic.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	abaaaa	babcabaa
4	cabaaa	ababcaba
5	acabaa	aababcab
6	bcabaa	caba
7	aaaaba	bcabaaca
8	aacaba	aaababca
9	abcaba	acaba
10	aaca	baaaabab
11	ababca	baacaba
12	baaaab	abcabaac
13	aaabab	cabaacab
14	aacab	aaaababc
15	babcab	aacaba
16	aac	abaaaaba
17	aababc	abaacaba

aacabaaaabab	cabaac	aba - - - - -

Shoda s aktuálním *kontextem cabaac* je ve slovníku na indexu 16. V tomto kroku se ovšem kódují poslední *symboly* řetězce, je tedy třeba vyhledávat aktuální *content* o 1 menší než je jeho velikost. Vyhledáním aktuální hodnoty *content ab* nalezneme první shodu od indexu 16 taktéž na indexu 16 a to ve dvou *symbolech*. Poslední trojice, která bude poslána na výstup je (16 - 16 = **0, 2, a**). Všechny *symboly* byly zakódovány, není tedy třeba se slovníkem či posuvným oknem dále zacházet.

Výstupním kódem jsou tyto trojice:

(0, 0, a), (0, 1, c), (-1, 1, b), (-3, 2, a), (5, 3, b), (-7, 5, c), (0, 2, a)

Tyto data je dále třeba binárně zakódovat. Kompresní poměr lze ilustrativně vypočítat takto: První složka trojice, která udává vzdálenost, má rozsah hodnot <-7, 5>, bude tedy třeba čtyř bitů při binárním kódování. Druhá složka má rozsah hodnot <0, 5>. Pro tuto složku budou třeba pouze tři bity. Poslední složka bude mít 8 bitů, protože bude použit standartní ASCII kód.

Celkovou velikost kódu můžeme tedy vypočítat jako $7 \cdot (4 + 3 + 8) = 105 \Rightarrow 112$ bitů oproti původním 21 *symbolů* $21 \cdot 8 = 168$ bitů. Kompresní poměr tedy vypočítáme jako $112/168 \doteq 0.667$. Ovšem v praxi se používají kódy s proměnlivou délkou slova, proto je tento výpočet pouze ilustrativní.

■

3.2 Dekomprese ACB

Dekompresi můžeme popsat pomocí krátkého pseudokódu zobrazeného v Algoritmu 2.

Algoritmus 2 ACB dekomprese

```
1: procedure ACBDecompress
2:     windowPosition ← 0                          ▷ nastavení pozice posuvného okna
3:     while windowPosition < InputSize do
4:         d ← načtení první složky (vzdálenost) ze zdrojového souboru
5:         l ← načtení druhé složky (počet shodných symbolů) ze zdrojového souboru
6:         s ← načtení třetí složky (symbol) ze zdrojového souboru
7:         i ← pozice nejlépe se shodujícího kontextu ve slovníku s aktuálním kontextem
8:         j ← i + d                          ▷ pozice shodné hodnoty content ve slovníku
9:         Output(j, l, s)                    ▷ zkopírování dekódované trojice do výstupu
10:        UpdateDictionary()   ▷ aktualizuj slovník (vlož nové fráze, aktualizuj hodnoty content)
11:        windowPosition ← windowPosition + l + 1
12:    end while
13: end procedure
```

Stejně jako při kompresi tak i při dekompresi má algoritmus na začátku prázdný slovník. V první fázi se přečte ze vstupního proudu první složka trojice (d), která udává vzdálenost mezi nejvíce se shodným *kontextem* a nejvíce se shodnou hodnotou *content*, druhou složku (*délka*) udávající počet shodných *symbolů* a poslední složku, která obsahuje první neshodný *symbol* (s).

Dalším krokem je nalezení původní shodné hodnoty *content*, který tentokrát nelze nalézt vyhledáním, protože při dekompresi je pravá část okna (aktuální *content*) neznámá, ale od toho tu je první složka (d). Nejprve je nalezen index (i) nejvíce se shodujícího *kontextu* ve slovníku podle aktuálního *kontextu*. K tomuto indexu (i) přičteme první získanou složku (d), z čehož dostaneme index (j) nejvíce se shodné hodnoty *content*.

Na výstup už tedy můžeme zapisovat dekódované *symboly*, které se nacházejí v hodnotě *content* na indexu (j) o počtu shodných *symbolů* druhé složky (l) s následným přidáním posledního *symbolu* (s). Poté se posuvné okno posune na další pozici.

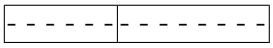
Ukázka dekomprese, pro lepší pochopení, bude dále uvedena v příkladu 2.

Příklad 2 (Dekomprese ACB)
Vstupními daty budou tyto trojice z příkladu 1:

$$I = \{ (0, 0, a), (0, 1, c), (-1, 1, b), (-3, 2, a), (5, 3, b), (-7, 5, a), (0, 2, a)\}.$$

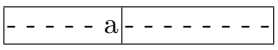
Velikost oken musí být nastavena stejně jako při kompresi. Velikost *kontextu* je tedy nastavena na hodnotu 6 a *content* na hodnotu 8. Pravá část okna (aktuální *content*) je v průběhu dekom-

prese neznámá. Po každém cyklu algoritmu je dekódováno jeden a více *symbolů*, které budou definovat aktuální *kontext*, který bude identický jako při kompresi.



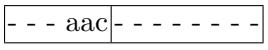
V prvním kroku, při trojici **(0, 0, a)** nám postačí pouze *symbol*, protože slovník je prázdný. Na výstup tedy je odeslán pouze *symbol* **a**. Slovník bude aktualizován o jednu položku. Posuvné okno se posune také o jednu pozici doprava.

index	*Kontext*	*Content*
0		a - - - - - - -



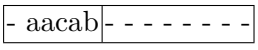
Další trojicí je **(0, 1, c)**. Pro aktuální *kontext* je vybrána jediná položka ve slovníku a to na indexu 0. Ve vzdálenosti **0** od této položky je vybrán **1** *symbol* z její hodnoty *content* a to *symbol a*. K tomuto *symbolu* je přidána třetí složka trojice neboli **c**. Výstupem je tedy dvojice *symbolů ac*. Slovník je rozšířen o dvě nové fráze a okno posunuto o dvě pozice doprava.

index	*Kontext*	*Content*
0		aac - - - - -
1	a	ac - - - - - -
2	aa	c - - - - - - -



V dalším kroku máme trojici **(-1, 1, b)**. Pro aktuální *kontext aac* je nalezen nejvhodnější *kontext* ve slovníku a to na indexu 2. Od tohoto indexu ve vzdálenosti **-1** se nachází *content ac*, z kterého potřebuje první **1** *symbol a*. K tomuto *symbolu* přidáme třetí složku z naší trojice neboli **b**. Na výstup je poslána dvojice *ab*. Ve slovníku přibyly dvě nové fráze se současným posunutím okna o dvě pozice.

index	*Kontext*	*Content*
0		aacab - - -
1	a	acab - - - -
2	aa	cab - - - - -
3	aaca	b - - - - - - -
4	aac	ab - - - - - -



Ze vstupu je načtena trojice **(-3, 2, a)**. Pro aktuální *kontext aacab* jsou nalezeny dva nejpodobnější *kontexty* ze slovníku a to na třetím a čtvrtém indexu. Při kompresi bylo již řečeno,

že se musí dodržovat stejná pravidla při výběru *kontextu*. Je tedy vybrán *kontext*, který je výše tedy na indexu 3. Ve vzdálenosti **-3** na indexu 0 je vybráno z hodnoty *content* první **2** *symboly*, ke kterým je přidán *symbol* **a**. Na výstup je odeslána trojice *symbolů aaa*. Slovník bude aktualizován o tři prvky a okno posunuto o tři pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaa -
2	aa	cabaaa - -
3	acabaa	a - - - - - - -
4	aacaba	aa - - - - - -
5	aaca	baaa - - - -
6	aacab	aaa - - - - -
7	aac	abaaa - - -

aa | cabaaa | - - - - - - - -

V dalším kroku pro trojici **(5, 3, b)** je nalezen *kontext* ve slovníku na indexu 2. Od tohoto indexu ve vzdálenosti **5** na indexu 7 je načten *content*. Z této hodnoty *content* jsou zkopírovány první **3** *symboly*, za které je přidán *symbol* **b**. Řetězec *abab* je odeslán na výstup. Slovník bude aktualizován o čtyři prvky a okno posunuto o čtyři pozice doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	abaaaa	bab
4	cabaaa	abab
5	acabaa	aabab
6	aaaaba	b
7	aacaba	aaabab
8	aaca	baaaabab
9	baaaab	ab
10	aacab	aaaabab
11	aac	abaaaaba

aacaba | aaabab | - - - - - - - -

Další trojicí je **(-7, 5, c)**. Pro aktuální *kontext aaabab* je nalezena shoda ve slovníku na indexu 9. K tomuto indexu je přičtena první složka **-7**, která identifikuje shodný *content*, který byl při kompresi, na indexu 2. Z hodnoty *content* na indexu 2 je vzato prvních **5** *symbolů*, ke kterým je přidán *symbol* **c**. Výsledný řetězec *cabaac* je odeslán na výstup. Do slovníku bude přidáno šest prvků a okno posunuto o šest pozic doprava.

index	*Kontext*	*Content*
0		aacabaaa
1	a	acabaaaa
2	aa	cabaaaab
3	abaaaa	babcabaa
4	cabaaa	ababcaba
5	acabaa	aababcab
6	bcabaa	c
7	aaaaba	bcabaac
8	aacaba	aaababca
9	abcaba	ac
10	aaca	baaaabab
11	ababca	baac
12	baaaab	abcabaac
13	aaabab	cabaac
14	aacab	aaaababc
15	babcab	aac
16	aac	abaaaaba
17	aababc	abaac

aacabaaaabab | cabaac | - - - - - - - -

Poslední trojicí je **(0, 2, a)**. Shoda *kontextů* ve slovníku je na indexu 16, který vlastní i shodný *content*, protože první složka trojice obsahuje hodnotu **0**. Shoda hodnoty *content* je na prefixu prvních **2** *symbolů*, ke kterým je přidána třetí složka **a**. *Symboly*, které budou odeslány na výstup, jsou *aba*. Slovník ani posuvné okno není třeba dále měnit.

Výsledkem výstupu po dekompresi je řetězec *aacabaaaababcabaacaba*, který se shoduje s řetězcem T z kompresního příkladu 1. ■

3.3 Modifikace algoritmu

Tato část práce se zaměřuje na možné modifikace algoritmu, které by mohli zlepšit výslednou kompresi.

3.3.1 Snížení redundance výstupního kódu

Při zapisování výsledných dat na výstup jsou případy, kdy dvě složky z trojice jsou nadbytečné a to v případech kdy druhá část trojice (*délka*) obsahuje hodnotu **0**. V takovém případě je hodnota první části (*offset*) irelevantní.
Tento problém lze řešit jedním z následujících způsobů:

1. Jako první se zapíše hodnota druhé části trojice (*délka*) a poté podle její velikosti se do souboru zapíše jedna z možností:
    - *délka = 0* - na výstup bude zapsána hodnota *délky* a poté pouze *symbol*

- *délka > 0* - na výstup bude zapsána hodnota *délky*, *offset* a poté *symbol*

2. Před každým prvkem z dat bude zapsán jeden bit, který bude udávat, zda následující prvek z dat bude složen ze tří částí (o, l, s) nebo pouze z jedné (s). Význam hodnoty bitu může být tedy následující:

   - *bit = 0* - na výstup bude zapsána pouze hodnota *symbolu*
   - *bit = 1* - na výstup bude zapsána hodnota *offsetu*, *délky* a poté *symbolu*

3.3.2 Rozdělení dat na dva typy

Při získávání dat (*trojic*) je možné, že není třeba zakódovat třetí složku (*s*) každý cyklus. *Symbol* je potřeba mít hlavně v případech, kdy jej není možné nalézt ve slovníku nebo pokud během vyhledávání byl výsledek druhé složky trojice příliš malý. V takovém případě by mělo být lepší zapsat pouze *symbol* (**s**) nebo pouze nalezenou shodu **(o, l)**. Rozdělení je tedy podle hodnoty druhé části (*délka*):

- délka >= minimální délka - k výstupním datům je přidána dvojice **(o, l)**
- délka < minimální délka - k výstupním datům je přidán pouze *symbol* **(s)**

Hodnota *délky* v datech se tedy dá využít jako identifikátor daného typu.

4 Implementace algoritmu ACB

Pro implementaci algoritmu ACB jsem zvolil programovací jazyk C#, jelikož jej znám nejlépe.

4.1 Datové struktury

4.1.1 Vstupní data

Vstupní data jsou načtena jako předem definované pole o statické velikosti typu bajt.

4.1.2 Výstupní data (trojice)

Pro výstupní trojice je vytvořena jednoduchá struktura, která se skládá ze tří složek: *offset*, *délka* a *symbol*.

4.1.3 Slovník

Slovník se skládá z obousměrného spojového seznamu, který je určen pro lineární vyhledávání, jež se využívá při hledání shodné hodnoty *content*. Dále se slovník skládá z binárního vyhledávacího stromu, který je určen pro snadné vyhledávání a vkládání prvků pro zmíněný seznam. Prvkem tohoto stromu je tedy reference na uzel z obousměrného spojového seznamu.

Ve slovníku je implementována fronta, která je určena pro naposledy přidané uzly binárního stromu. Při zavolání konstruktoru slovníku je definován maximální počet položek pro daný slovník. Pokud by byla tato hodnota při vkládání nového prvku překročena, je z fronty vybrán nejstarší uzel, který je následně odstraněn jak z binárního stromu tak i z obousměrného spojového seznamu.

Slovník obsahuje čtyři veřejné metody, které jsou využívány při kompresi i dekompresi. Vkládání nových prvků do slovníku je zprostředkováno metodou

void Insert(DictionaryElement input)

ve které je hlídán maximální počet prvků ve slovníku. Během dekomprese jsou využívány dvě metody. První metodou je

LinkedListNode<DictionaryElement> findContext(DictionaryElement currentElement)

, která slouží pro nalezení ideálního *kontextu* pro aktuální *kontext*. Druhá metoda se nazývá

LinkedListNode<DictionaryElement> findContent(LinkedListNode<DictionaryElement> context, int offset)

a slouží pro nalezení hodnoty *content* ve vzdálenosti od určitého *kontextu*. Další a poslední metodou, kterou tento slovník využívá, je metoda

int findMatch(DictionaryElement currentElement, int radius, out int outputLength)

, která je využívána během komprese. Tato metoda je ve své podstatě opakem předešlé metody, určené pro vyhledávání hodnoty *content*, protože jejím cílem je získat vzdálenost ideální hodnoty *content* od vybraného *kontextu* v daném *rádiu*, která bude následně uložena jako první část výsledné trojice. Z tohoto důvodu tato metoda musí využívat stejnou metodu pro nalezení *kontextu*. Důležité je také získat počet shodných *symbolů* pro druhou složku trojice, který je předán jako výstupní parametr.

4.1.4 Položka slovníku

Položka slovníku je implementována jako třída, která se skládá z reference na samotná data, z indexů, jež odkazují na *kontext* a *content* v daných datech a z hodnot označující velikosti *kontextu* a hodnoty *content*. Tato třída dědí z rozhraní *IComparable* s následnou definicí metody *CompareTo*, která se využívá při porovnávání jednotlivých prvků. Porovnávání probíhá od posledních *symbolů kontextů* obou porovnávaných prvků.

4.1.5 Kodér a dekodér

Kodér a dekodér jsou implementovány jako samostatná statická třída, která pomocí veřejné metody

public static void CompressFile(string fileName, int contextSize, int contentSize, int radius, int dictionarySize)

zkomprimuje soubor na dané absolutní cestě z parametru *fileName* s definovaným nastaveními ostatních parametrů. Pro dekompresi takto vytvořeného souboru je definována metoda s názvem

public static void DecompressFile(string sourcePath)

ve které je pouze předána absolutní cesta k souboru s příponou ACB. Původní soubor je poté vytvořen na místě, kde se nachází komprimovaný soubor.

V těchto veřejných metodách je využita privátní metoda

private static List<ACBData> getACBData(byte[] input, int contextSize, int contentSize, int radius, int dictionarySize)

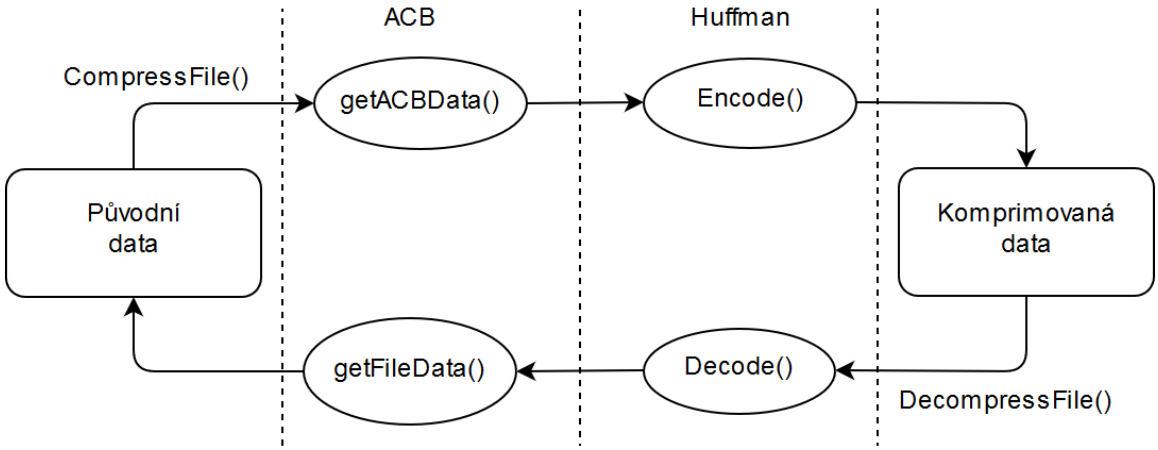
definovaná dle algoritmu 1 z pododdílu 3.1, jejíž účelem je získání dat typu *ACBData*. Pro získání původních dat dle algoritmu 2 z pododdílu 3.2 se používá metoda

private static List<byte> getFileData(ACBData[] input, int contextSize, int contentSize, int dictionarySize).

Tyto metody vytváří jednotlivé prvky slovníku, které jsou do něho následně ukládány. S následným využitím metod slovníku nalezne vyhovující hodnoty *content* a *kontexty*, z kterých vytváří data (*trojice*) pro výstup.

Ve veřejných metodách je dále využita veřejná třída *Huffman*, která používá metody *Encode* a *Decode*, jež jsou popsány v pododdílu 4.2 (Kódování výstupu).

Na obrázku 2 je jednoduché znázornění daných metod ze tříd ACB a Huffman pro kompresi a dekompresi souboru.



Obrázek 2: Jednoduché znázornění průběhu veřejných metod třídy *ACB*

Všechny uvedené metody v tomto pododdílu jsou dále modifikovány a jejich název je lehce upraven podle daných modifikací algoritmu, které budou dále popsány níže v pododdílu 4.3.

4.2 Kódování výstupu

Jednotlivé trojice jsou kódovány pomocí statického Huffmanova kódování ve statické třídě pojmenované Huffman. Pro Huffmanovo kódování je implementován binární zapisovač, který slouží pro zapisování primitivních typů v binárním kódu do datového proudu s přesností na jeden bit.

Huffmanovo kódování je implementováno ve statické třídě, která pro kódování využívá metodu

public static void Encode(string filename, ACBData[] data, int contextSize, int contentSize, int dictionarySize).

První parametr této metody slouží pro určení místa, kde bude nový soubor s příponou *ACB* uložen, další parametry určující velikosti *kontextu*, hodnoty *content* a slovníku, které byly použity při vytváření dat a poté samotná data, která jsou připravena pro zakódování.
Data jsou v této metodě kódována klasickým způsobem v tomto pořadí: *offset*, *délka* a *symbol*.

Pro dekódování je využita metoda

public static ACBData[] Decode(string fileName, out string extension, out int contextSize, out int contentSize, out int dictionarySize).

Podle prvního parametru je načten soubor, ve kterém je uložena přípona (*extension*) původního souboru, počet dat, velikost *kontextu*, hodnoty *content*, slovníku a také metoda komprese. Ná-

vratovou hodnotou této metody jsou data, která budou následně použita v kodéru pro obnovení původních dat.

4.3 Modifikace

Tato část práce popisuje implementaci dvou zmíněných modifikací z pododdílu 3.3. Modifikace byly implementovány jako nové metody s rozdílným názvem. Rozdělení dat na dva typy se zejména týká úprav soukromých metod třídy *ACB* a modifikace za účelem snížení redundance výstupního kódu se zejména týká úprav metod ve třídě *Huffman*.

4.3.1 Modifikace třídy ACB

Klasické získávání dat metodou *getACBData()* bylo přejmenováno na *getFileACBDataClassic()*. Modifikace z pododdílu 3.3.2 byla realizována v soukromých metodách třídy *ACB*. V této modifikaci jsou data dělena na dva typy podle hodnoty *délky*. Název této metody je getACBData2Types(). Veřejné metody byly upraveny tak, aby mohli využívat klasickou či modifikovanou metodu této třídy a také modifikované metody třídy *Huffman*, které jsou popsány níže.

4.3.2 Modifikace třídy Huffman

Klasické kódování dat metodou *Encode()* bylo přejmenováno na *EncodeClassic()*. V implementaci třídy *Huffman* byla realizována modifikace zmíněná v pododdílu 3.3.1, kde počet složek jednotlivých dat, určených k zakódování, je měněn v závislosti na hodnotě druhé složky trojice (*délka*). První způsob pro použití dané modifikace je pojmenován v metodách jako *EncodeLengthFirst()*, kdy je první zapisována *délka* a pokud je větší než nula, následuje za ní *offset* se *symbolem*, jinak pouze *symbol*. Druhý způsob je pojmenován v metodě jako EncodeBitFirst(), ve které je, při hodnotě bitu nula, zapsán pouze *symbol*, jinak celá trojice. Jak je možné vidět z diagramu tříd na obrázku 3, tak se veřejné metody, které slouží pro kódování a dekódování získaných ACB dat dělí podle modifikace využité jak ve třídě *ACB* tak ve třídě *Huffman*.

4.4 Parametry metody

4.4.1 Hlavní parametry

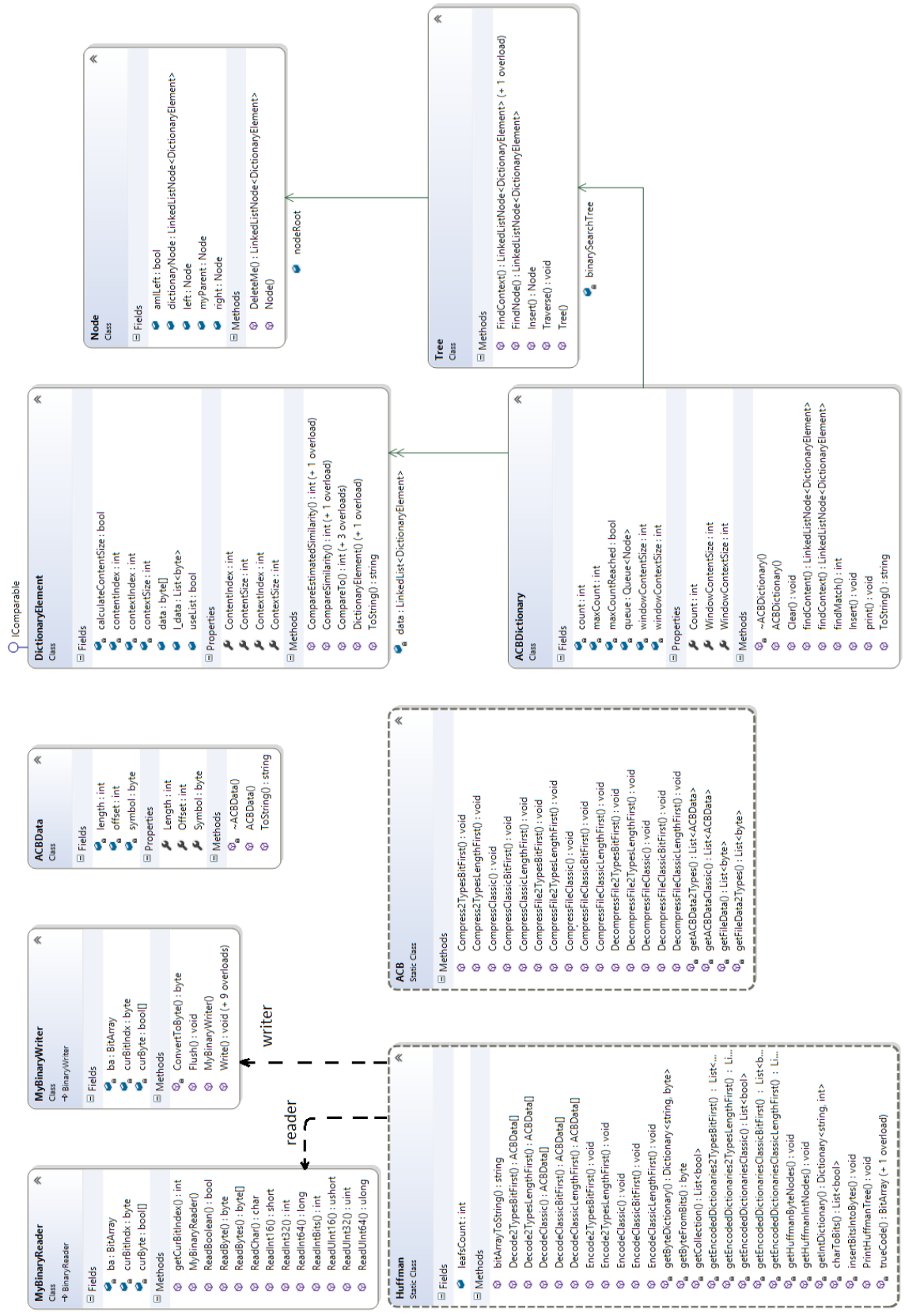
- **contextSize** - Hodnota tohoto parametru udává maximální délku *kontextu* v počtu bajtů.
- **contentSize** - Hodnota tohoto parametru udává maximální délku hodnoty *content* v počtu bajtů.
- **radius** - Hodnota tohoto parametru udává maximální velikost *offsetu* v bitech.
- **dictionarySize** - Hodnota tohoto parametru udává maximální počet prvků ve slovníku.

4.4.2 Parametry v modifikacích

- **minimumLength** - Hodnota tohoto parametru udává minimální *délku* potřebnou pro zakódování u modifikace getACBData2Types().

4.5 Diagram tříd

Diagram tříd je zobrazen na obrázku 3. Hlavní třídou implementace je třída *ACB*, která definuje veřejné metody pro kompresi a dekompresi, ve kterých využívá své privátní metody pro převod dat typu bajt na data typu *ACBData* a naopak.
V těchto privátních metodách je použit slovník *ACBDictionary*, který využívá obousměrný spojový seznam s hodnotami typu *DictionaryElement*, které reprezentují jednotlivé dvojice hodnoty *content* a *kontextu*. Tento slovník také využívá binární vyhledávací strom typu *Tree*, který obsahuje hlavní uzel (kořen) typu *Node* disponující odkazy na levý a pravý uzel (podstromy). Tyto uzly mají v sobě uloženu hodnotu typu *LinkedListNode<DictionaryElement>* což je reference na uzel obousměrného spojového seznamu typu *DictionaryElement*.
Po využití těchto privátních metod je nutno získaná data zakódovat. Pokud byly data typu *bajt*, tak se tyto data jednoduše uloží jako soubor získaného typu při dekódování. V případě, že tyto data jsou typu *ACBData* je třeba je zakódovat pomocí veřejných metod statické třídy *Huffman*, která díky svým privátním metodám vytvoří bitové reprezentace pro jednotlivé hodnoty *offsetu*, *délky* a *symbolu* podle jejich četnosti. Data jsou poté uloženy v souboru s příponou *ACB* za použití binárního zapisovače *MyBinaryWriter*.



Obrázek 3: Diagram tříd

4.6 Použití

Tato část popisuje použití všech implementovaných metod s možností měnit jejich jednotlivé parametry. Program je ve formě konzolové aplikace a je ovládán pomocí parametrů příkazové řádky. Při spuštění programu bez argumentů nebo s argumenty *–help*, *--h* nebo */?* je vypsána následující nápověda pro použití programu:

```
$./acb --help
Použití:
  acb <type> <filePath> <method> <contextSize> <contentSize> <searchRadius>
  <dictionarySize> <minimumLength>
Argumenty:
 type           Jaký typ operace má být proveden.
                   0 - Komprese
                   1 - Dekomprese
 filePath       Absolutní cesta k souboru, který má být komprimován nebo
                dekomprimován.
 method         (Pouze komprese) Hlavní metoda, která bude použita ke kompresi
                souboru.
                   0 - Klasická metoda
                   1 - ClassicLength metoda
                   2 - ClassicBit metoda
                   3 - 2TypesLength metoda
                   4 - 2TypesBit metoda
 contextSize    (Pouze komprese) Velikost kontextu v bajtech.
 contentSize    (Pouze komprese) Velikost části content v bajtech.
 searchRadius   (Pouze komprese) Vyhledávací rádius v bitech.
 dictionarySize (Pouze komprese) Maximální počet prvků ve slovníku.
 minimumLength  (Pouze komprese u metod 2Types) Minimální délka,
                která má být kódována.
Možnosti:
 -h, --help, /?  Zobrazí tuto nápovědu.
Nápověda:
        Tento program je implementací kompresní metody ACB. Program umožňuje
 kompresi a dekompresi vybraného souboru  s umožněním změny jednotlivých
 parametrů a implementovaných metod.
Příklady:
 acb 0 C:\bible.txt 0 64 1024 10 65536
 acb 1 C:\bible.ACB
```

Tato verze nápovědy je v českém jazyce. Pokud by byla zjištěna jiná lokalizace systému, v kterém je tento program spuštěn, zobrazí se nápověda v jazyce anglickém.

5 Výsledky měření implementovaným algoritmem

Tato část práce se zaměřuje na posouzení výsledků naměřeným při kompresi testovacích souborů. Mezi testované soubory byl zařazen korpus *Calgary*, který se od roku 1990 stal v praxi standardem pro bezeztrátové vyhodnocování komprese, tento korpus obsahuje osmnáct různých souborů. Tento a více korpusů je možno najít na webových stránkách "*canterbury*" [8]. Další soubory přidané do testování jsou: textový soubor mého programu s názvem **ACB** ve formátu **cs** a binární soubor **sum** z korpusu *cantrbry*. Podrobnější informace o použitých souborech je možné nalézt v příloze A.

Pro každé měření v této kapitole byla realizována kontrola, zda je obsah původního souboru shodný s obsahem nově dekomprimovaného souboru.

Program byl spouštěn v operačním systému Windows 8.1 Pro (x64) s procesorem Intel Core i5-6600 s operační pamětí 16 GB.

5.1 Vliv základních parametrů na kvalitu komprese

Tato část je zaměřena na porovnání závislosti jednotlivých parametrů klasicky implementované metody na výsledné velikosti zkomprimovaného souboru a rychlosti komprese. V této části bude využita skupina těchto čtyř souborů: **geo** (velikost 102 400 bajtů), **ACB.cs** (velikost 81 810 bajtů), **sum** (velikost 38 240 bajtů) a **book1** (velikost 768 771 bajtů).

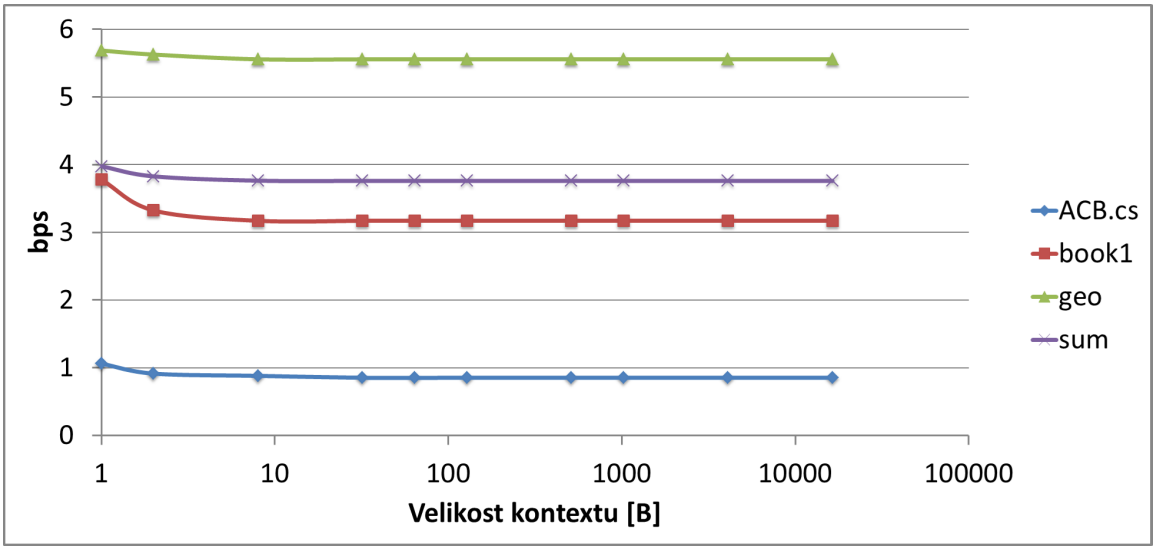
Pro všechny případy měření, v této podkapitole, byly nastaveny tyto výchozí hodnoty parametrů: velikost *kontextu* 64 bajtů, velikost hodnoty *content* 1 024 bajtů, velikost *slovníku* 65 536 prvků, velikost *offsetu* 10 bitů. Dále byl použit klasický algoritmus ACB s klasickým Huffmanovým kódováním trojic.

5.1.1 První parametr - velikost *kontextu*

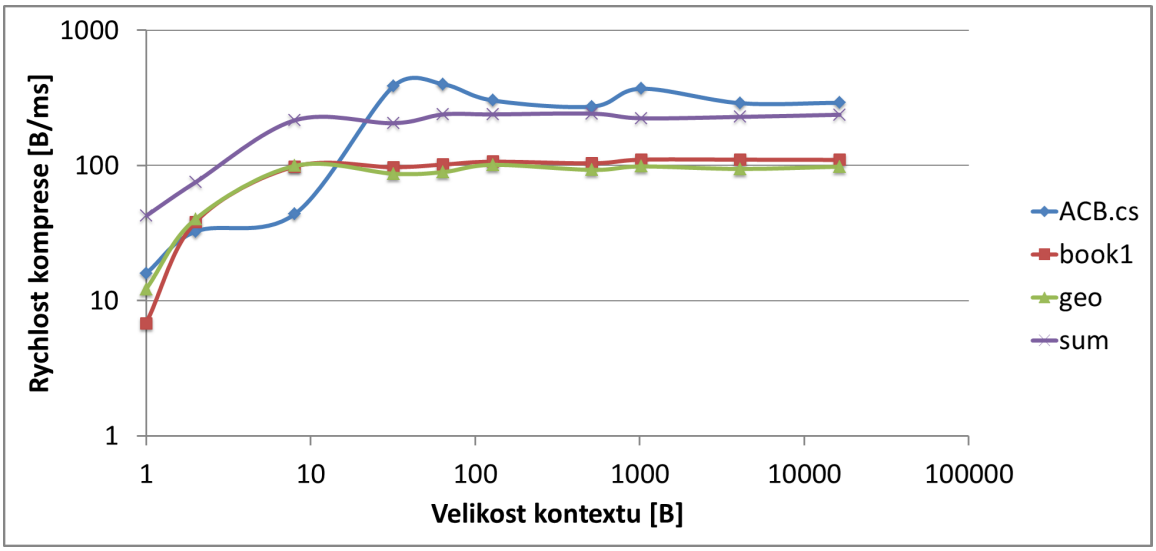
Velikost *kontextu* byla měněna od 1 bajtu do 16 384 bajtů. Obrázek 4 ukazuje závislost bps na změně velikosti *kontextu*. Z výsledků je patrné, že po dosažení velikosti osmi bajtů nemá velikost *kontextu* zásadní vliv na výslednou velikost komprimovaného souboru.

Obrázek 5 popisuje závislost rychlosti komprese (bajt za milisekundu) na změně velikosti *kontextu*. Od velikosti 64 se rychlost komprese stabilizuje a až na menší odchylky zůstává stejná.

Velice nízký *kontext* má ovšem negativní dopad na výpočetní rychlost. Hlavním důvodem tohoto zpomalení je s velkou pravděpodobností vznik mnoha duplicitních položek v implementovaném binárním vyhledávácím stromu, který při časté shodě *kontextu* vytváří jednosměrný list v levé části podstromu.



Obrázek 4: Závislost bps na velikosti *kontextu*

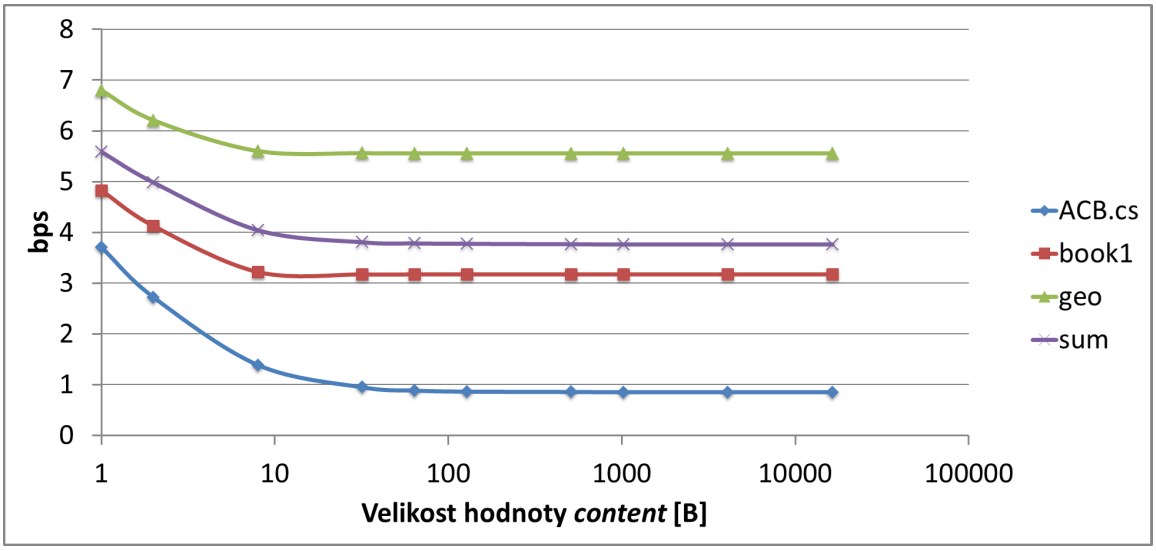


Obrázek 5: Závislost rychlosti komprese na velikosti *kontextu*

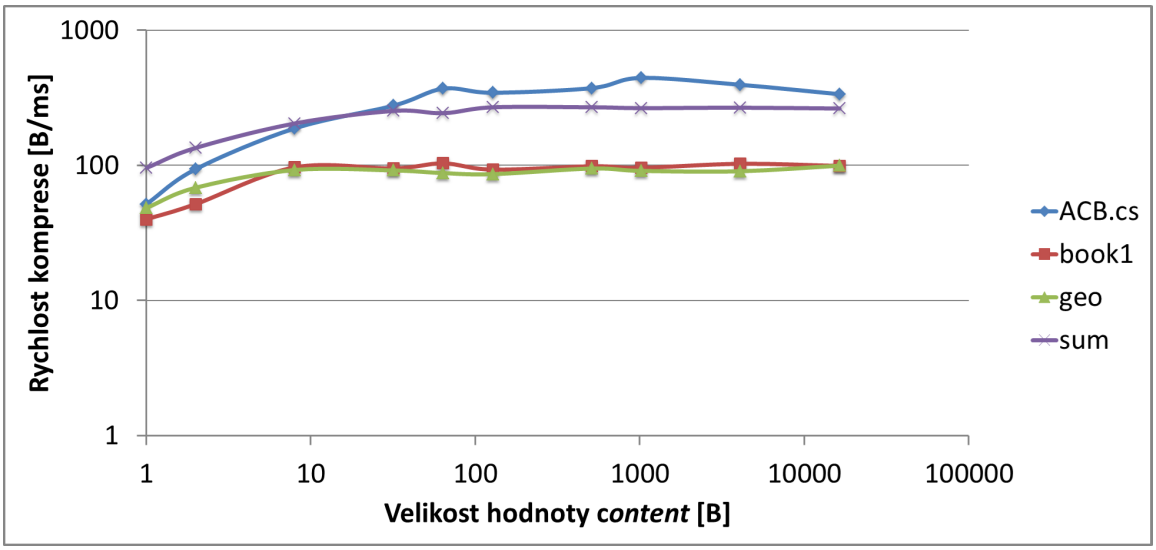
5.1.2 Druhý parametr - velikost hodnoty *content*

Při dalším měření byla měněna velikost hodnoty *content* od 1 bajtu do 16 384 bajtů. Závislost bps na změně velikosti hodnoty *content* je zobrazena na obrázku 6. Po dosažení velikosti 64 se kompresní poměr takřka nemění. Velikost hodnoty *content*, přestala plně ovlivňovat kompresní poměr až za hodnotou 1024.

Závislost rychlosti komprese (bajt za milisekundu) na změně velikosti hodnoty *content* je zobrazena na obrázku 7. Z grafu vyplývá, že rychlost komprese po dosažení velikosti hodnoty *content* 64 se nijak zásadně neliší.



Obrázek 6: Závislost bps na velikosti hodnoty *content*



Obrázek 7: Závislost rychlosti komprese na velikosti hodnoty *content*

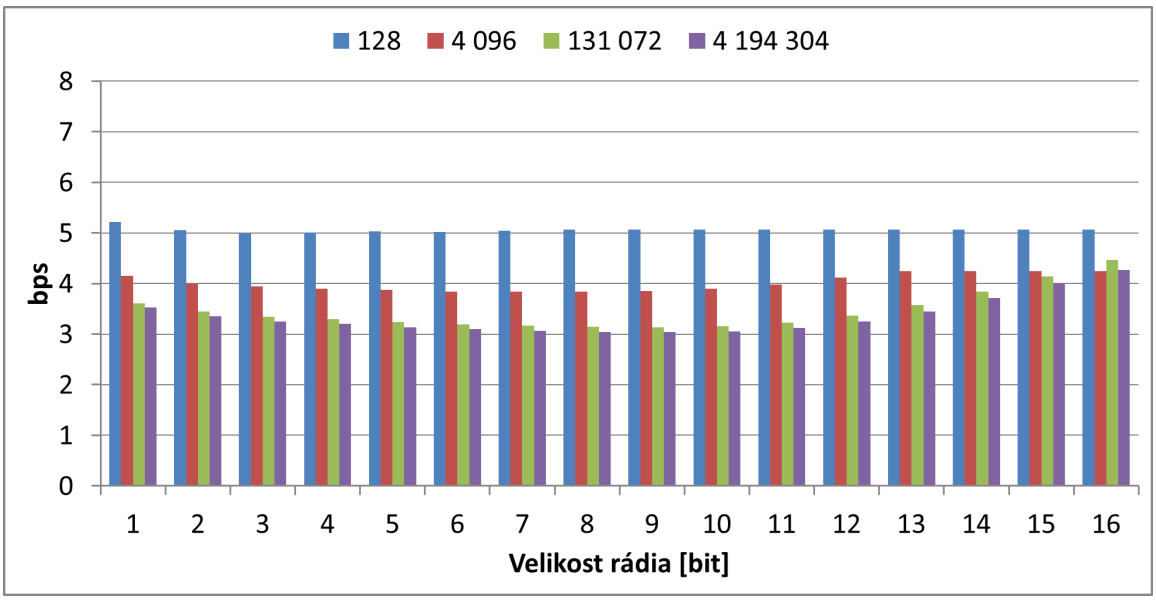
5.1.3 Třetí a čtvrtý parametr - velikost slovníku a *rádia*

Komprese byla použita na všechny čtyři soubory, ze kterých byl vypočítán vážený průměr počtu bitů na *symbol* a celková rychlost. Během měření byl měněn počet položek slovníku a velikost *rádia*, která ovlivňuje maximální hodnotu *offsetu*. Pro velikosti slovníku byly zvoleny hodnoty 128, 4 096, 131 072 a 4 194 304, které jsou v grafech rozděleny podle barvy. Velikosti *rádia* byla měněna od 1 bitu po 16 bitů, které jsou vyneseny na vodorovné ose.

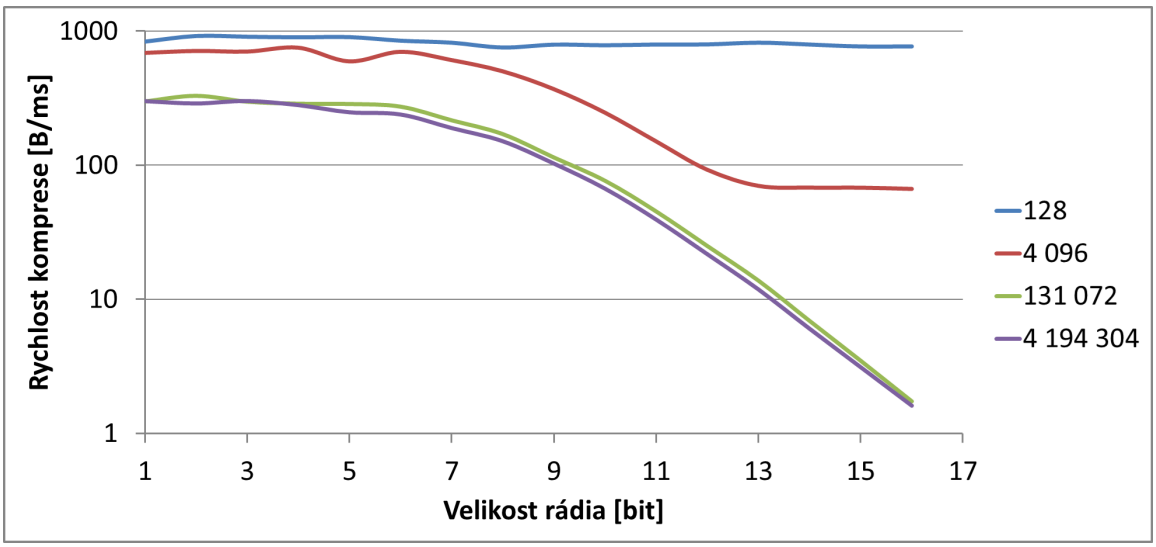
Na obrázku 8 je patrné, že největší slovník si vedl vzhledem ke kompresnímu poměru nejlépe. Dále lze vidět, že čím větší byl slovník, tím větší měla změna *rádia* vliv na výslednou kompresi. Oproti tomu příliš malý slovník nebyl takřka vůbec ovlivněn velikostí hodnoty *rádia*. Velikost

rádia přinášela nejlepší výsledky komprese od 6 bitů po 11 bitů. Příliš velký *rádius* způsoboval značné zhoršení komprese u velkého slovníku.

Obrázek 9 popisuje závislost rychlosti komprese (bajt za milisekundu) na změně velikosti *rádia* při různých velikostí slovníku. Během nízkých *rádiů*, byla komprese se slovníkem s počtem prvků 128 až třikrát rychlejší než komprese se slovníkem s počtem prvků 4 194 304. Po uskutečnění všech měření má právě velikost *rádia* největší vliv na výpočetní rychlost komprese, nesmí však být omezena velikostí slovníku. Rychlost komprese začíná nejvíce zpomalovat od hodnoty *rádia* s počtem 7 bitů.



Obrázek 8: Závislost váženého průměru bps na velikosti *rádia*



Obrázek 9: Závislost rychlosti komprese na velikosti *rádia*

5.1.4 Shrnutí vlivu základních parametrů na kvalitu komprese

Velikost *kontextu*, vzhledem k efektivitě komprese, by měla překračovat hodnotu 32 bajtů. Jako ideální hodnota bylo zvoleno 64 bajtů.

Pro velikost hodnoty *content* byla jako ideální hodnota zvoleno 1 024 bajtů. Tato hodnota má lepší a stabilnější výsledky počtu bitů na *symbol* bez negativního ovlivnění rychlosti.

Pro lepší kompresi by měl být zvolen větší slovník se středně velkým *rádiem*, který by se měl nejlépe pohybovat od 6 do 9 bitů. Jako ideální hodnota *rádia* je doporučeno 7 bitů. Tato hodnota má velice dobrý kompresní poměr se současným zachováním rychlosti komprese.

5.2 Rozložení četnosti výskytu offsetů a délky

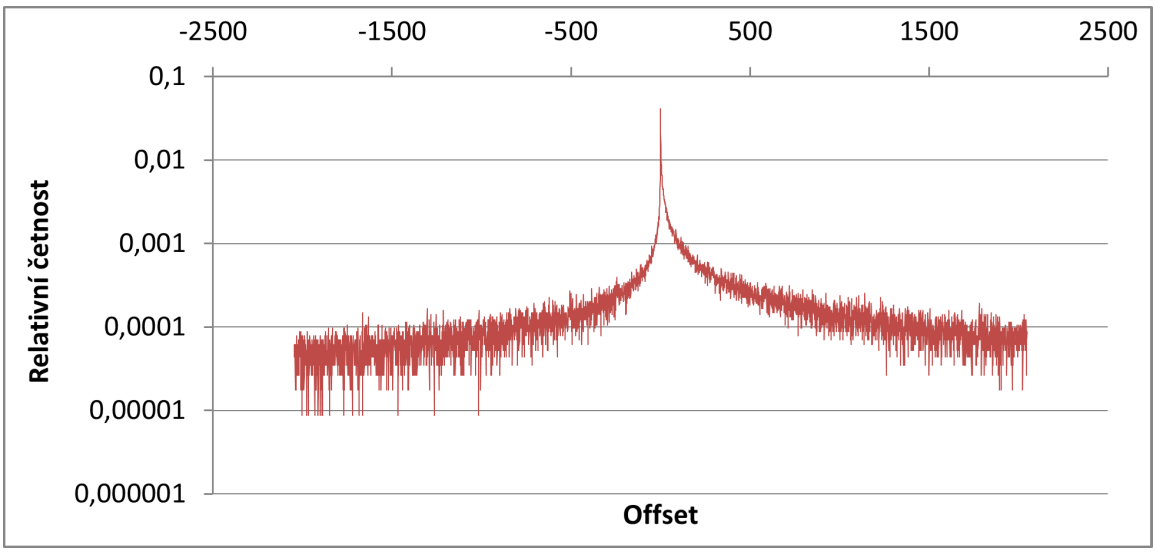
Měření četnosti výskytu *offsetů* a *délky* bylo uskutečněno na textovém souboru book1 z korpusu *Calgary*. Velikost *kontextu* byla nastavena na hodnotu 64 bajtů, velikost hodnoty *content* 1 024 bajtů, *rádius* vyhledávání 12 bitů (rozsah *offsetu* <-2 047; 2 048>) a maximální počet položek ve slovníku 4 194 304 (2^{22}).

Na obrázku 10 je zobrazena statistika reprezentující vztah relativní četnosti jednotlivých *offsetů* vzhledem k celkovému počtu výstupních dat (*trojic*) z klasické metody implementovaného algoritmu. Lze vidět, že se zvyšující se absolutní hodnotou *offsetu* se prudce snižuje jeho četnost.

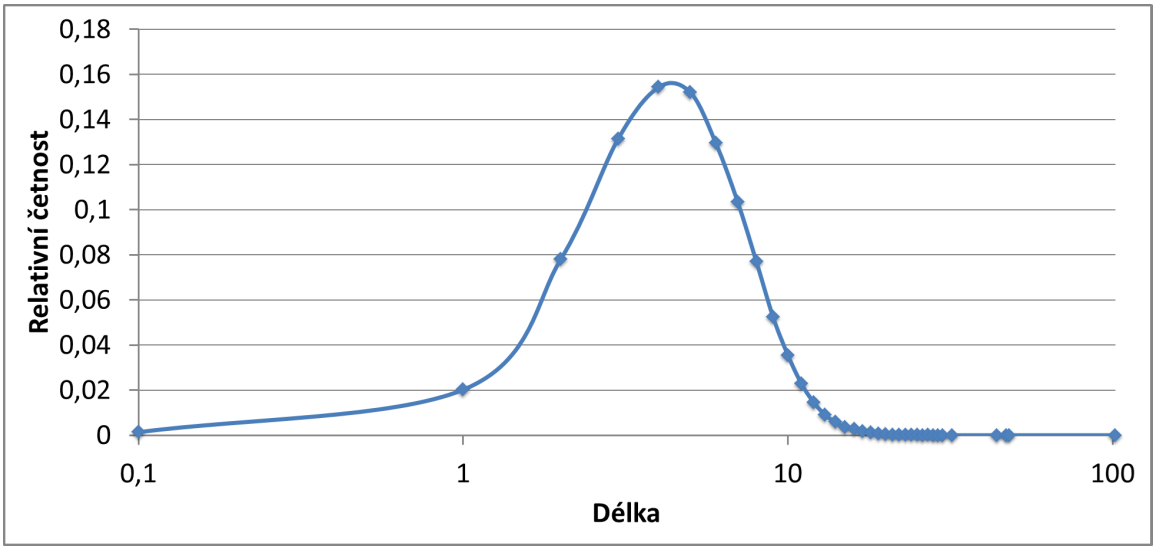
Obrázek 11 zobrazuje statistiku reprezentující vztah relativní četnosti jednotlivých *délek* vzhledem k celkovému počtu výstupních dat (*trojic*) z klasické metody implementovaného algoritmu. Nejvyšší četnosti dosahovala *délka* s hodnotou 4 a 5. Tato hodnota vyplývá z průměrné délky slova uvnitř zpracovaného textového souboru, která je rovna 4,19. Pokud přičteme k průměrné délce slova i mezeru dostáváme číslo 5,19. Tato hodnota se tedy velice blíží nejvyšší dosažené četnosti.

Rozložení *offsetů* a *délky* vyjadřuje tabulka 2, ve které je možno vidět, že 56,77% všech *offsetů* pochází z intervalu <-255; 256>. Interval hodnot *délky* <0; 15> obsahoval 99,5% výstupních dat (*trojic*).

Výsledky těchto statistik je možno vzít v úvahu při návrhu parametrů pro danou implementaci tohoto algoritmu.



Obrázek 10: Statistika četnosti hodnot offsetů u souboru book1



Obrázek 11: Statistika četnosti hodnot *délky* u souboru book1

Podíl[%]	Rozsah *offsetů*
15,56	<-2047; -256>
56,77	<-255; 256>
27,67	<257; 2048>

Podíl[%]	Rozsah hodnot *délky*
99,5	<0; 16>
0,43	<17; 21>
0,07	<22; 102>

Tabulka 2: Rozložení četnosti hodnot *délky* a *offsetu*

5.3 Vliv modifikací na kvalitu komprese

Tato část je zaměřena na porovnání závislosti klasické a modifikované implementace na výsledné velikosti komprimovaných souboru a jejich rychlosti komprese. V této části bude použit korpus *Calgary*.

Základní nastavení parametrů: velikost *kontextu* 64 bajtů, velikost hodnoty *content* 2 048 bajtů, maximální počet položek slovníku 4 194 304 a velikost *rádia* byla měněna od 4 do 10 bitů.

5.3.1 Použité metody komprese

Modifikace jsou rozděleny podle názvu, který reprezentuje druhy modifikací, jež byly použity pro metodu komprese. Názvy implementovaných metod použitých při tomto měření jsou:

1. **Classic** - Tento způsob používá klasické získávání dat algoritmem ACB, které následně zakóduje jako trojice pomocí klasického Huffmanova kódování.

2. **ClassicLength** - Tento způsob také používá klasické získávání dat algoritmem ACB. Ovšem následné Huffmanovo kódování trojic je provedeno prvním způsobem, který je popsán v pododdílu *"Snížení redundance výstupního kódu"* (3.3.1).
   Jako první je kódována *délka*, která při nulové hodnotě vynechá následující hodnotu *offsetu* z aktuální trojice.

3. **ClassicBit** - Opět je použito klasické získání dat algoritmem ACB. Tentokrát je ale použit způsobu druhý. Před každou trojici je identifikační bit, který definuje, zda je třeba číst pouze *symbol* nebo celá trojice.

4. **2TypesLength** - Tento způsob používá modifikované získávání dat algoritmem ACB popsaném v pododdílu *"Rozdělení dat na dva typy"* (3.3.2), které následně kóduje pomocí Huffmanova kódování, kde identifikátorem druhu dat je hodnota *délky*, která určuje, zda po ní následuje *offset* nebo *symbol*.

5. **2TypesBit** - Opět je použito získávání dat modifikovaným algoritmem ACB. Zápis dat během Huffmanova kódování je dělen pomocí jednoho bitu, který definuje, zda se bude číst pouze *symbol* nebo pouze *offset* a *délka*.

Dále pro porovnání byly použity metody komprese formátu ZIP a RAR za použití programu WinRAR ve verzi 5.31 sloužící pro archivaci souborů. Archivace byla nastavena na normální metodu komprese s velikostí slovníku 4 096KB(RAR) a 32KB(ZIP). Pro další porovnávání byly také vybrány některé metody komprese z webových stránek "*canterbury*" [8].

5.3.2 Srovnání výsledku různých druhů kompresí

Měření bylo provedeno s měnící se hodnotou *rádia* od 4 do 10 bitů. Pro každý soubor tedy bylo naměřeno 7 druhů výsledků, z kterých byla vybrána nejlepší komprese pro všechny metody. Minimální délka pro metody **2Types** byla nastavena na hodnotu 2.

Tabulka 3 ukazuje, jaký vliv měla metoda na kompresi daného souboru. Zvýrazněné buňky tabulky znázorňují nejnižší naměřenou hodnotu z implementovaných metod.

Nejlepších výsledků dosahovala kompresní metoda **2TypesBit**, která u velkých textových souborů jako je *book1*, *book2* a *news* dosahala i lepších výsledků než metoda ZIP.

Další tabulka s číslem 4 zobrazuje časy jednotlivých kompresí použitých v tabulce 3. Nejlepší časy byly naměřeny u metody **ClassicBit** a **2TypesBit**. Jak je možné vidět, metoda **2TypesBit** byla u komprese souborů *book1* a *book2* značně rychlejší než ostatní metody. Hlavním důvodem je to, že metodě **2TypesBit** stačilo pro lepší kompresi pouze *rádius* 7 bitů. Naproti tomu ostatní metody musely použít *rádiů* 9 až 10 bitů, aby získaly nejlepší kompresi. Průměrné hodnoty *rádia*, které byly použity pro získání nejlepší komprese u jednotlivých metod, jsou zobrazené v tabulce 5. Nejmenší hodnoty *rádia* byly nejčastěji použity v metodě **ClassicBit** což podle výsledků z tabulky 4 potvrzuje vliv hodnoty *rádia* na celkový čas komprese, která byla popsána v pododdíle 5.1.3.

Hodnoty, jakých nabýval rádius během nejlepších kompresí u jednotlivých souborů, jsou zobrazeny v tabulce 7 v příloze B.

	bps						
Soubor	Classic	ClassicLength	ClassicBit	2TypesLength	2TypesBit	ZIP	RAR
bib	2,509127	2,482235	2,565355	2,377473	2,377185	2,512147	2,372655
book1	2,917103	2,913752	3,071156	2,881607	2,943524	3,229019	2,883220
book2	2,497728	2,488783	2,612020	2,449180	2,477795	2,677331	2,372382
geo	5,513203	5,275625	5,275313	5,465391	5,465078	5,361641	4,875469
news	2,909779	2,884641	2,970579	2,785592	2,785508	3,050768	2,671822
obj1	4,376488	4,179315	4,177827	4,239583	4,238095	3,830357	3,631696
obj2	2,973413	2,905119	2,930758	2,741822	2,741692	2,626042	2,308265
paper1	2,976468	2,925453	2,988958	2,840729	2,839977	2,742763	2,724554
paper2	2,949720	2,926751	3,035852	2,868843	2,868453	2,851130	2,796628
paper3	3,268882	3,222456	3,300348	3,136139	3,135279	3,058419	3,054464
paper4	3,662803	3,553214	3,602589	3,493000	3,489989	3,296101	3,325606
paper5	3,748369	3,632592	3,663376	3,569684	3,567007	3,300653	3,326083
paper6	3,051778	2,985435	3,040231	2,896628	2,895788	2,747146	2,747776
pic	0,939550	0,920439	0,925957	0,904181	0,904118	0,816872	0,775720
progc	2,966247	2,897983	2,942819	2,820025	2,819217	2,657242	2,651385
progl	2,049968	2,010775	2,044273	1,938978	1,938531	1,783882	1,757418
progp	2,004091	1,968772	1,997286	1,923733	1,923085	1,798497	1,740011
trans	1,746347	1,710315	1,736443	1,651828	1,651486	1,598719	1,532632

Tabulka 3: bps vzhledem k použité metodě

	Čas komprese (ms)				
Soubor	Classic	ClassicLength	ClassicBit	2TypesLength	2TypesBit
bib	317	283	283	320	310
book1	12846	8067	12549	10547	5056
book2	5095	5222	5095	4505	4005
geo	643	534	555	516	466
news	2894	2039	1232	2163	1901
obj1	340	312	298	333	313
obj2	895	865	809	1523	1391
paper1	104	101	163	119	74
paper2	271	237	207	262	252
paper3	90	75	155	143	72
paper4	37	12	13	14	13
paper5	13	11	12	15	12
paper6	53	63	41	55	58
progc	81	72	52	60	59
progl	184	182	138	207	210
progp	57	89	83	79	64
trans	204	211	182	219	229
pic	1446346	1440964	1529817	1433421	1433317

Tabulka 4: Časy komprese vzhledem k použité metodě

	Průměrná hodnota rádia (bit)				
	Classic	ClassicLength	ClassicBit	2TypesLength	2TypesBit
rádius	6,11	5,61	4,72	5,94	5,77

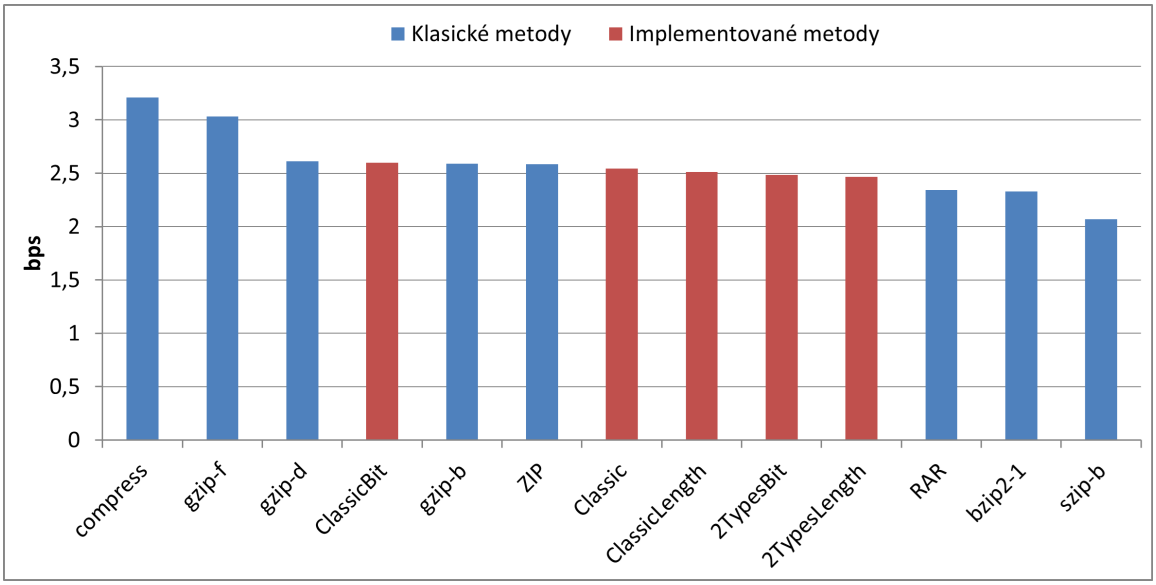
Tabulka 5: Průměrná velikost použité hodnoty rádia vzhledem k metodě.

Časy dekompresí modifikovaných metod jsou znázorněny v tabulce 6. Dekomprese souboru pic byla, vzhledem ke své časové náročnosti, vynechána. Všechny metody měly víceméně stejný počet nejlepších dosažených výsledků času dekomprese pro korpus *Calgary*.

Pro názornější porovnání modifikovaných metod mé implementace s klasickými metodami byl vytvořen graf na obrázku 12. Červenou barvou jsou vyznačeny mnou implementované metody komprese, barvou modrou jsou vyznačeny klasické komprese, které byly vybrány z tabulky kompresních výsledků na stránkách "*canterbury*" [8]. Na svislé ose je vynesena hodnota váženého průměru počtu bitů na *symbol*, který byl naměřen u všech souborů korpusu *Calgary*. Ovšem z měření mých implementovaných metod bylo třeba odebrat 4 soubory (paper3, paper4, paper5 a paper6), které už nejsou obsaženy v klasických metodách, protože nepřidávají nic nového k celkovému ohodnocení komprese.

	Čas dekomprese (ms)				
Soubor	Classic	ClassicLength	ClassicBit	2TypesLength	2TypesBit
bib	327	347	340	267	308
book1	4426	3903	4792	4634	3550
book2	2645	2650	2530	2739	2760
geo	395	410	393	634	419
news	1541	1510	1432	1538	1422
obj1	798	800	794	768	708
obj2	888	873	867	891	967
paper1	164	160	157	153	197
paper2	223	239	236	239	229
paper3	156	150	135	119	170
paper4	31	49	55	56	37
paper5	40	37	41	38	38
paper6	120	94	117	131	128
progc	84	91	135	132	126
progl	231	217	235	234	198
progp	151	127	150	118	151
trans	221	214	236	278	222

Tabulka 6: Časy dekomprese vzhledem k použité metodě



Obrázek 12: Závislost váženého průměru bps na zvolené metodě

5.3.3 Shrnutí vlivu modifikací na kvalitu komprese

Nejlépe si vedly metody **2Types**, které využívaly získávání dat modifikovaným algoritmem ACB oproti metodám **Classic**, v nichž byl použit klasický algoritmus. Co se týče následného Huffmanova kódování, vedla si lépe metoda *Length*, která pro rozpoznání typu dat používala hodnotu druhé složky (*délka*).

Ačkoliv metoda **2TypesBit** dosáhla více nejlepších kompresí v korpusu *Calgary*, metoda **2TypesLength** dosáhla, při zvážení průměru kompresních poměrů všech souborů, nejlepšího poměru komprese. Oproti původní implementaci, komprese metodou **2TypesLength** měla celkově až o 8% lepší výsledky. Největší rozdíly byly naměřeny u souborů *book1* a *book2*, u kterých metoda **2TypesLength** dosahovala značně lepších výsledků komprese než u ostatních souborů. S tímto faktem ovšem také klesla rychlost komprese u metody **2TypesLength**, která celkově jako druhá používala největší hodnoty *rádia*.
Metoda **ClassicBit**, která využívala při Huffmanově kódování jeden bit jako identifikátor vynechání *offsetu* byla ve více případech horší než původní metoda klasického ACB s klasickým Huffmanovým kódováním trojic. Jedná se tedy o modifikaci, která zhoršila výsledky komprese. Co se týče časové náročnosti, nejlepší celkové výsledky byly naměřeny u metod *2Types*. Značně pomalá komprese nastala u souboru s názvem *pic*, což je faxový soubor, jehož obsahem jsou hlavně bílé znaky. Celková dekomprese souborů *Calgary* zabrala zhruba 60% času oproti kompresi.

Pokud srovnáme výsledky kompresí modifikovaných metod s klasickými metodami komprese ze stránek "*canterbury*" [8] zjistíme, že například metody *Classic*, *ClassicLength*, *2TypesBit* a *2TypesLength* dosahovaly lepších výsledků než metody jako *gzip-b(d, f)* nebo *compress*. Dále tyto metody předčily svým kompresním poměrem i archivaci programem WinRar formátu ZIP.

6 Závěr

Cílem této bakalářské práce bylo navrhnout efektivní implementaci kompresní metody Associative Coder of Buyanovsky. Tato metoda byla po několika menších problémech úspěšně implementována. Jedním z hlavních problémů byl návrh datové struktury binárního vyhledávacího stromu, který je spojen s obousměrným spojovým seznamem spolu s následnou definicí metody pro odstranění určitého uzlu tohoto stromu.

Práce byla realizována v jazyce C# v prostředí Microsoft Visual Studio 2015. Mé předešlé zkušenosti v tomto jazyce mi byly v začátcích přínosem a během realizace této práce se v mnohém rozšířily. V práci byly použity vlastní datové struktury vybrané pro rychlou a efektivní práci algoritmu.

Podrobný popis algoritmu ACB je obsažen ve 3 kapitole práce, před kterou následuje krátký úvod. Po popsání algoritmu ACB následuje návrh možných modifikací, které byly v kapitole 4 implementovány s následným návodem pro jejich použití v programu.

Algoritmus byl testován nad různými testovacími soubory a také porovnán s jinými algoritmy. Pomocí změn základních parametrů metody byl zjištěn jejich vliv na výslednou kompresi a rychlost. Potěšujícím výsledkem mé práce bylo zjištění, že se mi částečně podařilo vhodně implementovat algoritmus ACB s tím, že jeho výsledky komprese předčily kompresi programu WinRAR ve formátu ZIP. Výsledky mé implementace algoritmu byly navíc zlepšeny pomocí modifikace, která dosáhla lepších výsledku komprese než původní implementace.

Podařilo se mi dosáhnout výsledků, které mi pomohli pochopit praktický význam jednotlivých parametrů algoritmu ACB. Návrhy možných modifikací by mohly v budoucnosti pomoci s dalším výzkumem tohoto algoritmu.

Literatura

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression", IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, 1977.

[2] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, 1978.

[3] S. Wolfram, "Notes for Chapter 10: Processes of Perception and Analysis: Data Compression", in A new kind of science: notes from the book, Champaign, IL: Wolfram Media, c2002, p. 1069.

[4] G. Buyanovsky: Associative Coding(v Ruštině), Monitor, Moscow, #8, 10-19, August 1994.

[5] D. Salomon, Data compression: the complete reference, 4th ed. London: Springer, c2007.

[6] M. Simpson, S. Biswas, and R. Barua, Analysis of Compression Algorithms for Program Data. University of Maryland, 2003.

[7] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, 1952.

[8] "The canterbury corpus", 2000. [Online]. Dostupné: http://corpus.canterbury.ac.nz/. [Přistoupeno: 14-Mar.-2018].

A Seznam testovacích souborů

Tato tabulka obsahuje seznam souborů, které byly použity při testování v této práci.

Soubor	Typ	Velikost (B)	Popis	zdroj
ACB	cs	81 810	Soubor třídy programu ACB	Vlastní
bib	-	111 261	Bibliografie	Calgary
book1	-	768 771	fiktivní kniha	Calgary
book2	-	610 856	skutečná kniha	Calgary
geo	-	102 400	Geofyzikální data	Calgary
news	-	377 109	USENET dávkový soubor	Calgary
obj1	-	21 504	Kód objektu pro VAX	Calgary
obj2	-	246 814	Objektový kód pro Apple Mac	Calgary
paper1	-	53 161	Technický papír	Calgary
paper2	-	82 199	Technický papír	Calgary
paper3	-	46 526	Technický papír	Calgary
paper4	-	13 286	Technický papír	Calgary
paper5	-	11 954	Technický papír	Calgary
paper6	-	38 105	Technický papír	Calgary
pic	-	513 216	Černobílý faxový obrázek	Calgary
progc	-	39 611	Zdrojový kód v "C"	Calgary
progl	-	71 646	Zdrojový kód v LISP	Calgary
progp	-	49 379	Zdrojový kód v PASCAL	Calgary
sum	-	38 240	Spouštěcí program SPARC	Canterbury
trans	-	93 695	Přepis terminálové relace	Calgary

B Použitý rádius k dosažení nejlepší komprese pro daný soubor

	Použitý *rádius* vyhledávání (bit)				
Soubor	Classic	ClassicLength	ClassicBit	2TypesLength	2TypesBit
bib	7	6	5	6	6
book1	10	9	10	9	7
book2	9	9	9	8	7
geo	4	4	4	6	6
news	9	8	4	7	7
obj1	4	4	4	5	5
obj2	6	6	5	8	8
paper1	6	5	4	5	5
paper2	7	7	4	6	6
paper3	6	5	4	5	5
paper4	4	4	4	4	4
paper5	4	4	4	5	5
paper6	6	4	4	5	5
pic	6	6	4	6	6
progc	6	5	4	5	5
progl	6	5	4	6	6
progp	5	5	4	5	5
trans	5	5	4	6	6

Tabulka 7: Použitý *rádius* k dosažení nejlepší komprese pro daný soubor

C Obsah přiloženého CD

Obsahem přiloženého CD je zdrojový kód mé implementace ACB algoritmu, samostatný zkompilovaný program ACB, testovací soubory a kopie této práce ve formátu PDF a LaTeX.